RESTifying Real-World Systems: a Practical Case Study in RFID

Dominique Guinard, Mathias Mueller, Vlad Trifa

Abstract As networked sensors become increasingly connected to the Internet, RFID or barcode-tagged objects are likely to follow the same trend. The EPC Network is a set of standards to build a global network for such electronically tagged goods and objects. Amongst these standards, the Electronic Product Code Information Service (EPCIS) specifies interfaces to capture and query RFID events from external applications. The query interface, implemented via SOAP-based Web services, enables business applications to consume and share data beyond companies borders and forms a global network of independent EPCIS instances. However, the interface limits the application space to the rather powerful platforms which understand WS-* Web services. In this chapter we introduce tools and patterns for Web-enabling real-world information systems advertising WS-* interfaces. We describe our approach to seamlessly integrate RFID information systems into the Web by designing a RESTful (Representational State Transfer) architecture for the EPCIS. In our solution, each query, tagged object, location or RFID reader gets a unique URL that can be linked to, exchanged in emails, browsed for, bookmarked, etc. Additionally, this enables Web languages such as HTML and JavaScript to directly use RFID data to fast-prototype light-weight applications such as mobile applications or Web mashups. We illustrate these benefits by describing a JavaScript mashup platform that integrates with various several services on the Web (e.g., Twitter, Wikipedia,

Dominique Guinard
ETH Zurich, Switzerland, MIT Auto-ID Labs, USA
and SAP Research, Switzerland
e-mail: dguinard@ethz.ch

Mathias Mueller
Software Engineering Group, University of Fribourg, Switzerland

Vlad Trifa
Institute for Pervasive Computing, ETH Zurich, Switzerland

The original publication is available at www.springerlink.com published in the book: REST: From Research to Practice, edited by E. Wilde and C. Pautasso.

etc.) with RFID data to allow managers along the supply chain and customers to get comprehensive data about their products.

1 Introduction

The EPC Network is composed of several standards addressing issues ranging from the RFID (Radio Frequency Identification) tags themselves (EPC standard) to readers infrastructure and the reading middleware [3]. These standards define how to encode, read and aggregate data about tagged objects throughout the whole supply chain. Furthermore, to be able to query and use recorded RFID data (i.e., traces), the EPCIS standard (Electronic Product Code Information Services) acts as a global track and trace sharing infrastructure with several, potentially interconnected, EPCIS servers distributed around the world. The EPCIS provides a simple and lightweight HTTP interface for recording EPC events. A different approach is taken to querying for these traces by other applications because the EPCIS specifies a standardized WS-* (i.e., SOAP, WSDL, etc.) interface. The WS-* integration type has been successfully used to combine business applications [9, 10]. For example, it can be used to integrate EPCIS data about the status of a shipment with an *Enterprise Resource Planning (ERP)* application.

However, WS-* applications are complex systems with a high entry barrier as it requires developer expertise in the domain. Hence, they are not optimal for more lightweight and ad-hoc application scenarios [9]. Furthermore, the WS-* protocols are known to be rather verbose. Moreover, they do not fully meet the requirements of resource-constrained devices such as mobile phones and wireless sensor/actuator networks often not providing WS-* server or even client stacks [12, 8]. As a consequence, these shortcomings limit the type of applications built on top of EPCIS servers to rather heavy-weight business applications fully supporting the WS-* protocols. This is unfortunate since track and trace applications are also relevant beyond the desktop. As an example, providing an out-of-the-box mobile access to this data might be beneficial for many users such as mobile workers. Similarly, providing direct access to RFID traces to sensor and actuator networks could enable those to react to RFID events. Finally, allowing lightweight Web applications (e.g., HTML, JavaScript, PHP, etc.) to directly access these data would enable the vast community of Web developers to create innovative applications using RFID traces.

In this chapter, we illustrate how a RESTful API (Application Programming Interface) for the EPCIS opens new design possibilities for RFID applications. First, it **lowers the entry barrier** for developers and fosters rapid prototyping. Second, it enables **direct access** to RFID data without any additional software other than the EPCIS itself. Direct access to EPC events, allows to read, test, bookmarked, exchange, share RFID-related data from any Web browser, a tool that is ubiquitously available and understood by a vast number of people [7]. Finally, it enables a more **lightweight** access to the data. This is particularly desirable for applications that need to access EPCIS data from **resource-constrained devices** such as mobile

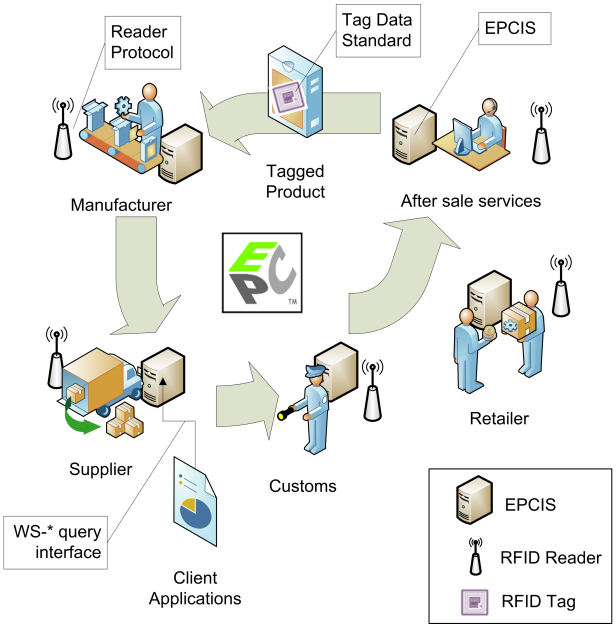

Fig. 1 Simplified view of the EPC Network and some of its main standards

phones or sensor nodes. REST is known to be more light-weight [12] than WS-* services and many resource-constrained devices are REST-ready through simple HTTP client libraries or higher-level REST client libraries.

The chapter is structured as a "cookbook" each section begins with some theoretical background (recipe) and is then applied (cooked) to the implementation of the RESTful EPCIS. We start by briefly presenting the REST constraints. We then propose two implementation patterns and describe tools that can greatly speed up the development process of a RESTful enterprise system. We finally, illustrate how REST fosters the "mashability" of real-world information systems with the EPC Mashup Dashboard. This Web mashup platform allows the exploration of EPC-related data and gathering of timely information about tagged objects from various Web services such as Twitter, Wikipedia or Google Maps. Product or supply chain managers can use this tool as a business intelligence platform to better understand and visualize the entire supply chain. Likewise, customers can better understand and visualize where different come from, what other people think about them, and so on.

Before looking at the "RESTification" process, we briefly introduce the EPC Network and summarize the basic concepts behind RESTful Web Services.

1.1 An Introduction to the EPC Global Network

As illustrated on Figure 1, the EPC Network[1] is a set of standards established by industrial key players towards a uniform platform for tracking and discovering RFID tagged objects and goods. Fifteen standards are currently composing the EPC Network addressing every step required from encoding data on RFID tags to reading them and sharing their traces. We will focus on two of them as those are the most relevant in the context of this paper.

The first standard is the EPC Tag Data Standard (TDS). It defines what an EPC number is and how it is encoded on the tags themselves as shown on the product box of Figure 1. An EPC is a world wide unique number. Rather than identifying a product class, like most barcode standards do, it can be used to identify the instance of a product. The TDS specifies eight encoding schemes for EPC tags. They basically contain three types of information: the manufacturer, the product class and a serial number. As an example in the tag (represented in its URI form): urn:epc:id:gid:2808.64085.88828, `2808` is the manufacturer ID, `64085` represents the type of product and `88828` an instance of the product.

One of the goals of the EPC Network is to allow sharing observed EPC traces. Thus, the network specifies a standardized server-side EPCIS, in charge of managing and offering access to traces of EPCs events. Whenever a tag is read it goes through a filtering process and is eventually stored in an EPCIS together with contextual data. In particular, these data deliver information about:

- The “what”: what tagged products (EPCs) were read.
- The “when”: at what time the products were read.
- The “where”: where the products were read, in terms of Business Location (e.g., “Floor B”).
- The “who”: what readers (Read Point) recorded this trace.
- The “which”: what was the business context (Business Step) recording the trace (e.g., “Shipping”).

The goal of the EPCIS is to store these data to allow creating a global network where participants can gain a shared view of these EPC traces. As such, the EPCIS deals with historical data, allowing, for example, participants in a supply chain to share the business data produced by their EPC-tagged objects.

Technically speaking, a standard EPCIS is an application that offers three core features to client applications:

1. First it offers a way to capture, i.e., persist, EPC events.
2. Then, it offers an interface to query for EPC events.
3. Finally, it allows to subscribe to queries so that client applications can be informed whenever the result of a query changes.

There exist several concrete implementations of EPCISs on the market. Most of them are delivered by big software vendors such as IBM or SAP. However, the Fos-

[1] http://epcglobalinc.org/standards/architecture

strak [3] project offers a comprehensive, Java-based, open-source implementation of the EPCIS standard.

The great potential of the EPC network for researchers in the ubiquitous computing field has led to a number of initiatives trying to make it more accessible and open for prototyping than it currently is. The authors of [3] initiated the Fosstrak project, which is to date the most comprehensive open-source implementation of the EPC standards. The Fosstrak EPCIS is an open-source implementation of a fully-featured EPCIS. This project is suitable for prototyping [3] but it implements the standard WS-* interface which closes the EPCIS to a number of interesting use cases such as direct use from simple Web languages or usage on resource constrained devices.

To overcome these limitations, researchers started to create translation proxies between the EPCIS and their applications. In [6] the authors present an implementation of such a proxy. The "Mobile IoT Toolkit" offers a Java servlet based solution that allows to request some EPCIS data using URLs which are then translated by a proxy into WS-* calls. This solution is a step towards our goal as it enables resource-constrained clients such as mobile phones to access some data without the need for using WS-* libraries. Nevertheless, the proxy is directly built on the core of Fosstrak and thus does not offer a generic solution for all EPCIS compliant system. Furthermore, the protocol used in this implementation as well as the data format is proprietary which requires developers to learn it first.

In the "REST Binding" project[2] a translation proxy is implemented, similarly to [6] it proposes using URLs for accessing the EPCIS data but these data are provided using the XML format specified in the standard. While this is an important improvement, the proposed protocol does not respect the REST principles but implements what experts sometimes call a REST-RPC style [11]. As we will explain in the next Section, the connectedness and uniform interface properties do not held. Thus, an EPCIS using this interface is not truly integrated to the Web [10, 11]. To better understand this, let us summarize some of the core notions of RESTful Web Services.

2 RESTful Information Systems

REST is an architectural style, which means that it is not a specific set of technologies. For this paper, we focus on the specific technologies that implement the Web as a RESTful system, and we propose how these can be applied to the Web of Things. The central idea of REST revolves around the notion of resource as *any component of an application that needs to be used or addressed*. Resources can include physical objects (e.g., a temperature sensors, an RFID tagged object, etc.) abstract concepts such as collections of objects, but also dynamic and transient concepts such as server-side state or transactions. REST can be described in five constraints:

[2] http://autoidlabs.mit.edu/CS/content/OpenSource.aspx

- *Resource Identification:* the Web relies on *Uniform Resource Identifiers (URI)* to identify resources, thus links to resources can be established using a well-known identification scheme.
- *Connectedness: (also known as: Hypermedia Driving Application State* Clients of RESTful services are supposed to follow links they find in resources to interact with services. This allows clients to "explore" a service without the need for dedicated discovery formats, and it allows clients to use standardized identifiers and a well-defined media type discovery process for their exploration of services. This constraint must be backed by resource representations (having well-defined ways in which they expose links that can be followed.
- *Uniform Interface:* Resources should be available through a uniform interface with well-defined interaction semantics, as is *Hypertext Transfer Protocol (HTTP)*. HTTP has a very small set of methods `GET`, `PUT`, `POST`, and `DELETE` with different semantics (*safe*, *idempotent*, and others), which allows interactions to be effectively optimized.
- *Self-Describing Messages:* Agreed-upon resource representation formats make it much easier for a decentralized system of clients and servers to interact without the need for individual negotiations. On the Web, media type support in HTTP and the *Hypertext Markup Language (HTML)* allow peers to cooperate without individual agreements. For machine-oriented services, media types such as the *Extensible Markup Language (XML)* and *JavaScript Object Notation (JSON)* have gained widespread support across services and client platforms. JSON is a lightweight alternative to XML that is widely used in Web 2.0 applications and directly parsable into JavaScript objects.
- *Stateless Interactions:* This requires requests from clients to be self-contained, in the sense that all information to serve the request must be part of the request. HTTP implements this constraint because it has no concept beyond the request/response interaction pattern; there is no concept of HTTP sessions or transactions.

The design goals of RESTful systems and their advantages for a decentralized and massive-scale service system align well the field of pervasive computing: millions to billions of available resources and loosely coupled clients, with potentially millions of concurrent interactions with one service provider. Based on these observations, we argue that RESTful architectures are the most effective solution for the global Web of Things [5], composed of smart appliances, sensor nodes and tagged objects. Indeed these architectures scale better and are more robust than RPC-based architectures like WS-* services.

2.1 Case Study: RESTifying the EPC Information Service

As mentioned before, in the EPCIS standard, most features are accessible through a WS-* interface. To specify the architecture of the RESTful EPCIS we systemat-

ically took these WS-* features and applied the properties of a *Resource Oriented Architecture (ROA)* we summarized in the previous section.

2.1.1 Resource Identification and Connectedness

All the services of a Resource Oriented Architecture are modeled with resources which are components of an application worth being uniquely addressed and linked to. Each resource gets a unique and resolvable address in the form of a URL. Thus, the first step a ROA design is to **identify the resources** an EPCIS should be composed of and to make them **addressable**. Looking at the EPCIS standard, we can extract a dozen resources. We focus here on the four main types:

1. Locations (called "Business locations" in the EPCIS standard): those are locations where events can occur, e.g.,:"C Floor, Building B72".
2. Readers (called "ReadPoints" in the standard): which are RFID readers registered in the EPCIS. Just as Business Locations, readers are usually represented as URIs: e.g., `urn:br:maxhavelaar:natal:shipyear:incoming` but can also be represented using free-form strings, e.g.,: "Reader Store Checkout"
3. Events: which are observations of RFID tags, at a Business Location by a specific reader at a particular time.
4. EPCs: which are Electronic Product Codes identifying products (e.g., `urn:epc:id:sgtin:618018.820712.2001`), types of products (e.g., `urn:epc:id:sgtin:618018.820712.*`) or companies (e.g., `urn:epc:id:sgtin:618018.*`).

We first define a hierarchical organization of resources based on the following URI template:
`location/businessLocation/reader/readPoint/time/eventTime/event`
More concretely, this means that the users begin by accessing the Location resources. Accessing the URL `http://.../location/` with the `GET` method retrieves a list of all Locations currently registered in the EPCIS. From there, clients can navigate to a particular Location where they will find a list of all Readers at this place. From the Readers clients get access to Time resources which root is listing all the Times at which Events occurred. By selecting a Time, the client finally accesses a list of Events.

Each event contains information such as its type, event time, Business Location, EPCs, etc. If a client is only interested about one specific field of an Event, he can get this information by adding the desired information name as sub-path of the Event URI. For example, `EVENT_URI/epcs` lists only all the EPCs that were part of that Event. The resulting tree structure is shown in Figure 2, and a sample Event in Figure 3.

Furthermore, in a ROA all resources should be discoverable by browsing to facilitate the integration with the Web. Just as you can browse for Web pages, we should be able to find RFID tagged objects and their traces by browsing. Each rep-

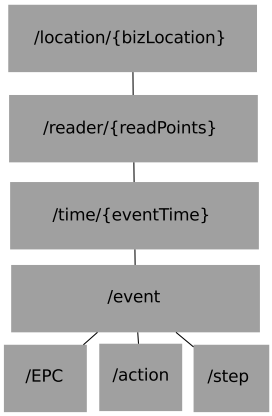

Fig. 2 Hierarchical representation of the browsable RESTful EPCIS resources

resentation of resources should contain links to relevant resources such as parents, descendants or simply related resources. This property of ROAs is known as "connectedness".

To ensure the connectedness of the RESTful EPCIS, each resource in the tree links to the resources below or to related resources. The links allow users to browse completely through the RESTful EPCIS where links act as the motor. Every available action is deduced by the set of links included. This way, people can directly explore the EPCIS from any Web browser, simply by clicking on hyperlinks and without requiring any prior knowledge of the EPCIS standard.

To ensure that the browsable EPCIS interface did not become too complicated, we limited the number of available resources and parameters. For more complex queries we provide a second, hierarchical, interface for which we map the EPCIS WS-* query interface to uniquely identifiable URIs. Each query parameter can be encoded and combined as a URI query parameter according to the following template

`/eventquery/result?param1=value1&...&paramN=valueN`

Query parameters restrict the deduced result set of matching RFID events. The RESTful EPCIS supports the building of such URIs with the help of an HTML form. If for example a product manager from Max Havelaar is interested in the events that were produced in Palmas, the following URL lists all events that occurred at this business location:

`http://.../eventquery/result?location=urn:br:maxhavelaar:palmas:productionsite`

To further limit possibly very long search results, the query URI can be more specific. The manager might be interested only about what happened on that production site on the 4th of November 2009, which corresponds to the following URL:

`http:/../eventquery/result?location=urn:br:maxhavelaar:palmas:productionsite&time=2009-11-04T00:00:00.000Z,`

`2009-11-04T 23:59:59.000Z`
The HTML representation of this resource is illustrated in Figure 3.

To keep the full connectedness of the RESTful EPCIS, both the browsable and the query interface are interlinked. For example, the EPC `urn:epc:id:sgtin:0057000.123430.2025` included in the event of Figure 3, is also a link to the query which asks the EPCIS for all events that contain this EPC.

We leverage the addressability property to allow a greater interaction with EPCIS data on the Web. As an example, since queries are now encapsulated in URLs, we can simply bookmark them, exchange them in emails and consume them from JavaScript applications. Furthermore, by implementing the connectedness property we enable users to discover the EPCIS content in a simple, yet powerful manner.

2.1.2 Uniform Interface and Self-Describing Messages

Finally, in a ROA, the resources and their services should be accessible using a standard interface defining the mechanisms of interaction. The Web implementation of REST uses HTTP for this purpose.

Multiple Representation Formats A resource is representation agnostic and hence should offer several representations (e.g., XML, HTML). HTTP provides a way for clients to retrieve the most adapted one. The RESTful EPCIS supports multiple output formats to represent a resource. Each resource first offers an HTML representation as shown in Figure 3 which is used by default for Web browser clients.

In addition to the HTML representation, each resource has also an XML and a JSON (JavaScript Object Notation) representation, which all contain the same information. The XML representation complies with the EPCIS standard and is intended to be used mainly for business integration. The JSON representation can be directly translated to JavaScript objects and is thus intended for mashups, mobile applications or embedded computers.

The choice of the representation to use in the response can be requested by clients using the HTTP "content negotiation" mechanism[3]. Since content negotiation is built into the uniform interface, clients and servers have standardized ways to exchange information about available resource representations, and the negotiation allows clients and servers to choose the representation that fits best a given scenario.

A typical content negotiation procedure looks as follows. The client begins with a `GET` request on `http://.../location`. It also sets the `Accept` header of the HTTP request to a weighted list of media types it can understand, for example to: `application/json`, `application/xml;q=0.5`. The RESTful EPCIS then tries to serve the best possible format it knows about and describes it in the `Content-Type` of the HTTP response. In this case, it will serve the results in the JSON format as the client prefers it over XML (q=0.5).

[3] http://www.w3.org/Protocols/rfc2616/rfc2616-sec12.html

RESTful Path ID	ID	*Unique Path ID to represent the Event*		
Event Type	ObjectEvent			
Event Time	2009-11-04T10:21:39.000Z			
Time Zone Offset	+00:00			
Record Time	2010-02-26T15:04:59.000Z			
Business Location	urn:br:maxhavelaar:palmas:productionsite			
Read Point	urn:br:maxhavelaar:palmas:productionsite:outgoing			
Business Step	urn:epcglobal:epcis:bizstep:fmcg:shipping			
Action	OBSERVE			
EPC List				
EPC	urn:epc:id:sgtin:0057000.123430.2025	Company Prefix: 0057000	Item Reference: 123430	Serial Number: 2025
		Serialized Global Trade Item Number		
EPC	urn:epc:id:sgtin:0057000.123430.2026	Company Prefix: 0057000	Item Reference: 123430	Serial Number: 2026
		Serialized Global Trade Item Number		

Fig. 3 HTML representation of an EPC event as rendered by a Web browser, every entry is also a link to the sub-resources

Error Codes The EPCIS standard defines a number of exceptions that can occur while interacting with an EPCIS. HTTP offers a standard and universal way of communicating errors to clients by means of "status codes". Thus, to enable clients, especially machines to make use of the exceptions defined by the EPCIS specification, the RESTful EPCIS maps the exceptions to HTTP status codes. An exhaustive list of error codes and their meanings for Resource Oriented Architectures can be found in [11].

3 Syndication with Atom

In many cases, it would be useful to group tagged objects into collections according to certain properties or scenarios (example collections would be "all the milk bottles shipped today to rhode island" or "potatoes shipped to client no 3"), and be able to monitor the state of collection through a syndication mechanism. The Atom Syndication Format is an XML language specifying the syntax of Web feeds. With Atom, the Web has a standardized and RESTful model for interacting with collections, and the *Atom Publishing Protocol (AtomPub)* extends Atom's read-only interactions with methods for write access to collections. Because Atom is RESTful, interactions with Atom feeds can be based on simple `GET` operations which can then be cached.

3.1 Case Study: Web-Enabling the Subscriptions

Standard EPCISs also offers an interface to subscribe to RFID events. Through a WS-* operation, clients can send a query along with an endpoint (i.e., a URL) and subscribe for updates. Every time the result of the query changes, an XML packet containing the new results is sent to the endpoint. While this mechanism is practical, it requires for clients to run a server with a tailored Web applications that listens to the endpoint and thus cannot be used by all users or cannot be directly integrated to a Web browser.

This makes the subscription interface an ideal candidate to apply the idea of Web feeds with Atom. Thus, in the RESTful EPCIS, we propose an alternative Atom module for producing the results of query subscriptions as shown on the leftmost side of Figure 5. This way, end-users can formulate queries by browsing the RESTful EPCIS and get updates in the Atom format which most browsers can understand and directly subscribe to.

As an example a product manager could create a feed in order to be automatically notified in his browser or any feed reader whenever one of his products is ready to be shipped from the warehouse. More concretely, this means sending an HTTP `PUT` request to
`http://.../eventquery/subscription?reader=urn:ch:migros:stgallen:warehouse:expedition&epc=urn:epc:id:sgtin:0057000.123430.*`
Or, for a human client, clicking on the "subscribe' link present at the top of each HTML representation of query results. As a result, the RESTful EPCIS will create an Atom feed corresponding to this query and add an entry (using AtomPub) to the feed every time an event for the product category `123430` is generated by reader `urn:ch:migros:stgallen:warehouse:expedition`.

The product manager can then use the URI of the feed in order to send it to his most important customers, allowing them to follow the goods progress as well. A simple but very useful interaction which would require a dedicated client to be developed and installed by each customer in the case of the WS-* based EPCIS.

4 Implementing RESTful Information Systems

After the design of RESTful Services, comes their implementation. The recent interest for RESTful services has led to a number of frameworks helping developers in this step. In this section we will look at some of these frameworks, focusing on their features and benefits when applying the constraints of RESTful architectures. However, let us begin by looking at integration patterns at a higher level: given an existing information system, what integration options do we have?

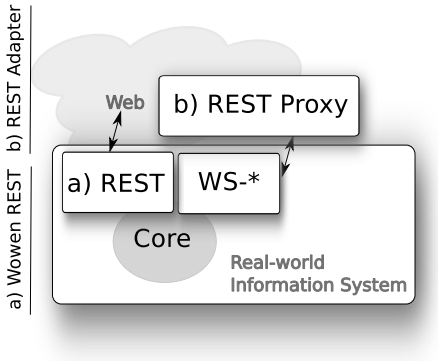

Fig. 4 Integration patterns for adding a RESTful interface to a WS-* system

4.1 From WS- to REST: Integration Patterns*

When creating an information system from scratch, the constraints for RESTful architectures are of great help in defining the data model. There are also no major conflicts between the REST paradigm and the Object Oriented paradigm. Indeed, Object Oriented programming defines an internal, application centric, contract. REST, on the other hand, defines a contract with the world outside the application (this is why developers often speak about RESTful APIs) towards a distributed and remote usage of its functionality. Thus, both can cohabit nicely to create a distributed Web application, as long as they are designed together. However, adding a RESTful architecture to an existing WS-* centric information system can be challenging as both paradigms share the same basic goal: creating remotely re-usable services.

Woven REST

As shown on Figure 4, there are basically two ways of achieving an integration; First (a) on Figure 4), the RESTful architecture can be directly woven into the existing WS-* system. This may seems like a trivial solution at first, however the implementation of this solution is not entirely straightforward. While sharing a common goal, WS-* and REST are rooted on very different paradigms. Thus, weaving clean REST architecture into the core of the WS-* system almost always requires an alternate data model. Having two data models for the same services ends up in rather complicated architectures.

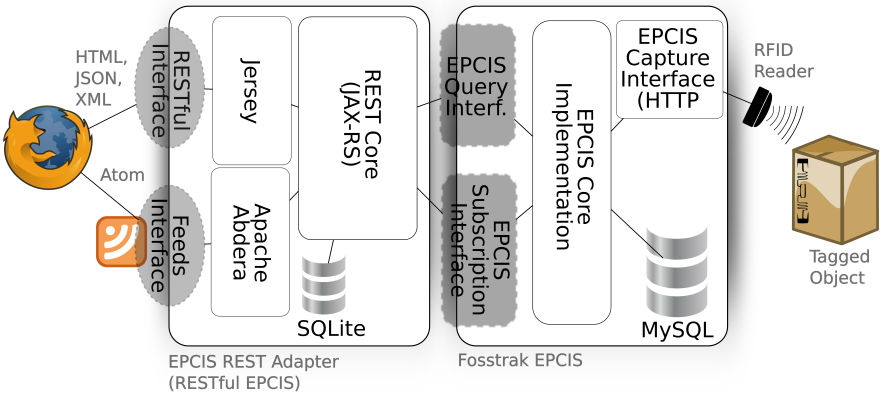

Fig. 5 Architecture of the RESTful EPCIS based on the Jersey RESTful framework and deployed on top of the Fosstrak EPCIS.

REST Adapter

An alternative integration pattern is to design an external REST Adapter making use of the WS-* interface, as shown in b) of Figure 4, REST Adapter. In this model, the REST Adapter acts as a proxy, translating RESTful requests into WS-* requests. This allows for a cleaner, REST centric architecture and preserves the legacy WS-* system entirely intact. On the downside it hinders the performances of the RESTful API but, as we will show in the case study, with a few simple measures, this can be minimized to a level acceptable for most applications.

4.1.1 Case-study: RESTful EPCIS as a Module

For the RESTful EPCIS, we decided to create an independent REST Adapter, as it delivers a clear advantage in this case: it allows the RESTful EPCIS to work on top of any standard EPCIS implementation.

The resulting architecture is shown in Figure 5. The RESTful EPCIS is a module which core is using the EPCIS WS-* standard interface. Just as a proxy, it translates the incoming RESTful request into WS-* requests and returns results complying with the constraints of RESTful architectures. As shown on the left of the picture, the typical clients of the RESTful EPCIS are different from the business applications traditionally connected to the EPCIS. The browser is the most prevalent of these clients. It can either directly access the data by means of URL calls or indirectly using scripted Web pages.

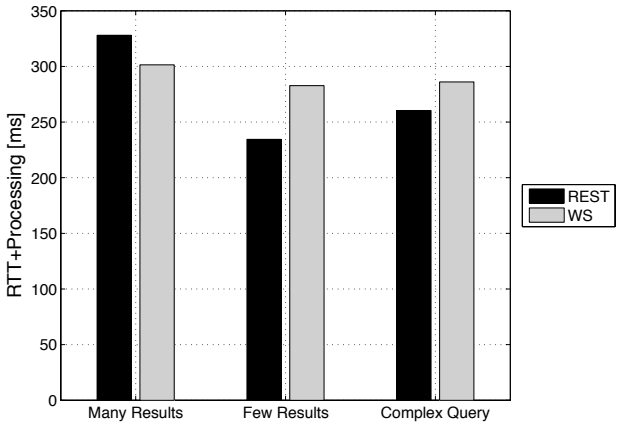

Fig. 6 Average RTT and processing time when using the WS-* interface and the REST interface for three types of requests each run 100 times.

Performance Evaluation

As mentioned before, the translation between REST and WS-* (and vice-versa) results in an overhead that we briefly evaluate here.

The experimental setup is composed of a Linux Ubuntu Intel dual-core PC 2.4 GHz with 2 GB of ram. We deploy Fosstrak and the RESTful EPCIS on the same instance of Apache Tomcat with a heap size of 512 MB. We evaluate three types of queries all returning the standard EPCIS XML representation.

The first query (Q1, "Many Results" in Figure 6) requests all events recorded by the EPC, i.e., a small request returning a document of 30 KB with 22 events each composed of about 10 EPCs. In the second test (Q2, "Few Results"), is a query returning a document of 2.2 KB with only two results. The last test (Q3, "Complex Query") is a query containing a lot of parameters and returning 10 events. We test each of these queries asking for the standard XML representation. All queries are repeated 100 times from a client located on a machine one hop away from the server with a Gigabit ethernet connectivity. The client application is programmed in Java and uses a standard JAX-WS client for the WS-* calls and the standard Apache HTTP Client and DOM (Document Object Model) library for the REST calls.

As shown on Figure 6, for Q1 the RESTful EPCIS has an average overhead of 30 ms due to the computational power required to translate the requests from REST to WS-* and vice-versa. For Q2 and Q3 the REST requests are executed slightly faster (about 20 ms) than the WS-*. This is explained by three factors. First, since there are fewer results, the local WS-* request from the RESTful EPCIS is executed faster. Then, REST packets are slightly smaller as there is no SOAP envelope [12]. Finally, unmarshalling WS-* packets (using JAXB) on the client-side takes significantly longer than for REST packets with DOM. For Q3, similar results are observed. Overall, we can observe that the RESTful EPCIS creates a limited overhead of about

10% which is compensated in most cases by the relatively longer processing times of WS-* replies. This becomes a particularly important point when considering devices with limited capabilities such as mobile phones or sensor nodes as well as for client-side (e.g., JavaScript) web applications.

It is worth mentioning that the WS-* protocol can be optimized in several ways to better perform, for example by compressing the SOAP packets and optimizing JAXB. However as the content of HTTP packets can also be compressed this is unlikely to drastically change the results. Furthermore, because they encapsulate requests in HTTP `POST`, WS-* services cannot be cached on the Web using standard mechanisms. For the RESTful EPCIS however, all the queries are formulated as HTTP `GET` requests and fully contained in the request URL. This allows to directly leverage from standard Web caching mechanisms [2] which would importantly reduce the response times [12].

4.2 Understanding the Tools Galaxy in Java

Creating clients for RESTful Web Services is a rather straightforward task as it only requires for the used language to support HTTP, which most modern programming and scripting languages do. The implementation of a RESTful Web Services, on the other hand, is a task that should not be underestimated. Indeed, even if the set of REST constraints is seemingly small their implementation requires a careful software design.

Most modern Web languages such as Ruby (especially in its Ruby on Rails form) or Python offer out-of-the-box support for RESTful Web Services. Similarly, the recent growing interest for lightweight service architectures based on REST has given birth to a number of frameworks that simplify the development of RESTful applications for enterprise-scale languages such as C# or Java.

4.2.1 JAX-RS: A Standard Java API for RESTful Web Services

The Java community is particularly interesting one since it is known as one of the community with most WS-* tools and frameworks but also as one of the most eager to develop tools around REST (perhaps due to some frustrations with the WS-* type of services...).

In particular, the Java galaxy has its own higher-level industrial standard for building RESTful Web Services: the JAX-RS API[4] (also known as JSR 311). JAX-RS is especially interesting since it was developed by a consortium of people who are both Web-specialists and service developers. The result is a very lean API (well described in [1]) that requires a good understanding of REST but offers straight-

[4] http://jcp.org/en/jsr/detail?id=311

forward solutions to implement in an elegant and efficient way most of the REST constraints.

In short, JAX-RS is based on three main pillars. It first uses annotations of Java classes to turn them into resources (e.g., `@Path(''/location'')`, ensuring the *Resource Identification* constraint. Annotations further help to define the resources' *Uniform Interface* as it lets the developer specify allowed verbs (`@GET, @POST`) and served representations (e.g., `@Produces(MediaType.APPLICATION_JSON)`). Beyond annotations, several framework classes make the developer life easier. *Connectedness* is boosted by providing contextual `URI Builders`, letting the developer easily link resources together across representation. Finally, the use of the JAXB framework allows for Java Objects to be automatically serialized to an (extensible) number of representations such as XML, HTML, JSON and Atom thus making it easier to fulfill the constraint for *Self-Describing Messages*.

Besides Jersey[5], the reference implementation of JAX-RS, several frameworks such as RESTeasy, Apache Wink, Apache CFX and RESTlet are JAX-RS compliant which makes it rather easy to move code from one framework to the other.

4.3 Case-study: Using JAX-RS, Jersey and Abdera

As shown in Figure 5, the core of the RESTful EPCIS is based on the JAX-RS compliant, Jersey[6] framework. Thus, it uses JAX-RS annotations and framework classes. The example below serves the representation of a `location` resource.

```
1   @Path(\location\{businessLocationID})
    @GET
3   @Produces({MediaType.APPLICATION_XML, MediaType.
        APPLICATION_JSON, MediaType.APPLICATION_ATOM_XML,
        MediaType.TEXT_HTML})
    public Resource getSelectedBusinessLocation(@Context
        UriInfo context, @PathParam("businessLocationID")
        String businessLocation) {
5       QueryBusinessLogic logic = new QueryBusinessLogic();
        return logic.getSelectedBusinessLocation(context,
            businessLocation);
7   }
```

Line 1 of this listing sets the URI of the resource, where `businessLocationID` is the `location` identifier which will be dynamically passed to the method `getSelectedBusinessLocation` at runtime. `@GET` specifies the method allowed on this resource, `@Produces` contains the representations that clients will be able to obtain through content negotiation. Note that these contents will be automatically generated at runtime from the `Resource` Java Object by the JAXB framework.

[5] http://https://jersey.dev.java.net

[6] https://jersey.dev.java.net

As we can see, the RESTful EPCIS uses Jersey for managing the resources' representations and dispatching HTTP requests to the right resource depending on the request URL. When correctly dispatched to the RESTful EPCIS Core, every request on the querying or browsing interface is then translated to a WS-* request on the EPCIS. This makes the RESTful EPCIS entirely decoupled from any particular implementation of an EPCIS.

While JAX-RS offers serving Atom representation of resources on-the-fly, implementations of JAX-RS do not have to offer a fully-featured Atom-Pub server with persistence. Thus, for the subscription interface we used Apache Abdera, which is an open-source implementation of an Atom-Pub server integrating well with most JAX-RS frameworks. Every time a client subscribes to a query, the RESTful EPCIS checks whether this feed already exists by checking the query parameters, in any order. If it is not the case it creates a query on the WS-* EPCIS and specifies the address of the newly created feed. As a consequence every update of the query is directly `POST`ed to the feed resource which creates a new entry using Abdera and stores it in an embedded SQLite[7] database.

Jersey, Abdera and SQLite are packaged with the RESTful EPCIS core in a WAR (Web Application Archive) that can be deployed in any Java compliant Web or Application Server. We tested it successfully on Glassfish[8] and Apache Tomcat[9] and on the Grizzly embedded Web Server[10].

5 REST and the Mashups

As RFID objects become part of the Web, applications using them can be developed using popular Web languages (e.g. HTML, JavaScript, PHP, Python) and toolkits, (e.g., DOJO, jQuery, Closure). This can significantly ease the developments on the RFID middleware vendor's side, since applications can be built on languages for which a plethora of libraries and toolkits are available. Furthermore, the use of popular languages makes it easier to find adequate developers. Likewise, this also unveils the possibility for external developers to create innovative Web applications making use of RFID data. Open APIs and communities of developers have long become vital for service companies on the Web such as Facebook, Twitter, or Google. This direction is also being taken upon by many electronic devices (sensor nodes, appliances, etc.). New hardware on the market such as the Chumby alarm clock[11] or the Squeezbox HiFi system[12] already have significant communities of voluntary Web developers creating dozens of small applications for each platform. Adding

[7] http://www.sqlite.org

[8] http://glassfish.org

[9] http://tomcat.apache.org

[10] http://grizzly.dev.java.net

[11] http://www.chumby.com

[12] http://www.logitechsqueezebox.com

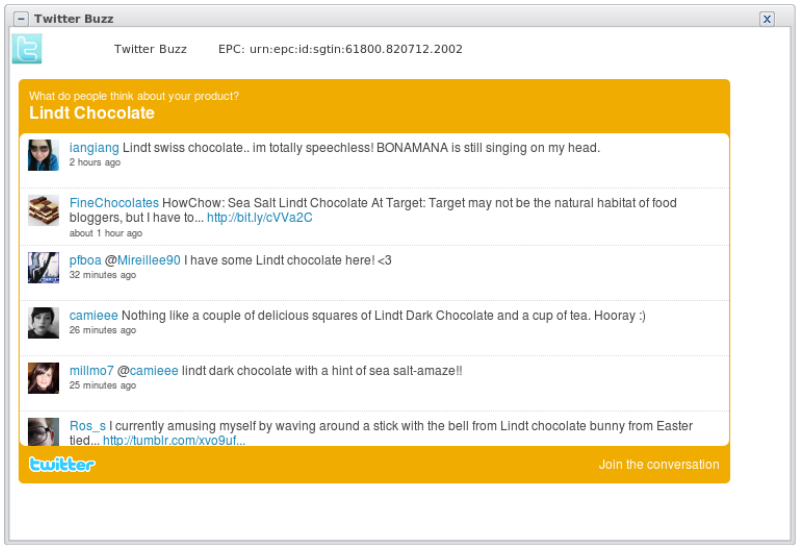

Fig. 7 The Product Buzz Widget extracts live opinions and information about particular products (here Lindt Chocolate) from Twitter

a RESTful module to the EPCIS brings it one step closer to these promising opportunities, where the consumers become active actors, not just passive consumers. Just as users create Web 2.0 mashups [13] by integrating several Web sites to create new applications, companies buying RFID systems can re-use RFID events to create ad-hoc, innovative applications in an easier manner. The EPCIS RESTful API allows a wider range of developers, tech-savvy users (technologically skilled people) or researchers to develop on top of the EPCIS and contributes to helping the EPC Network developer community grow.

5.1 Case Study: The EPC Dashboard Mashup

To better illustrate the new type of applications the RESTful EPCIS unveils we created the EPC Dashboard Mashup, a Web mashup, that helps product, supply chain and store managers to have a live overview of their business at a glance. It can further help consumers to better understand where the goods are coming from and what other people think about them. The EPC Dashboard is based on the concept of widgets in which the event data are visualized in a relational, spacial or temporal manner.

The EPC Dashboard consumes data from the RESTful EPCIS. Usually these data are hard to interpret and integrate. The dashboard makes it simple to browse and visualize the EPC data. Furthermore, it integrates the data with multiple sources on the Web such as Google Maps, Wikipedia, Twitter, etc.

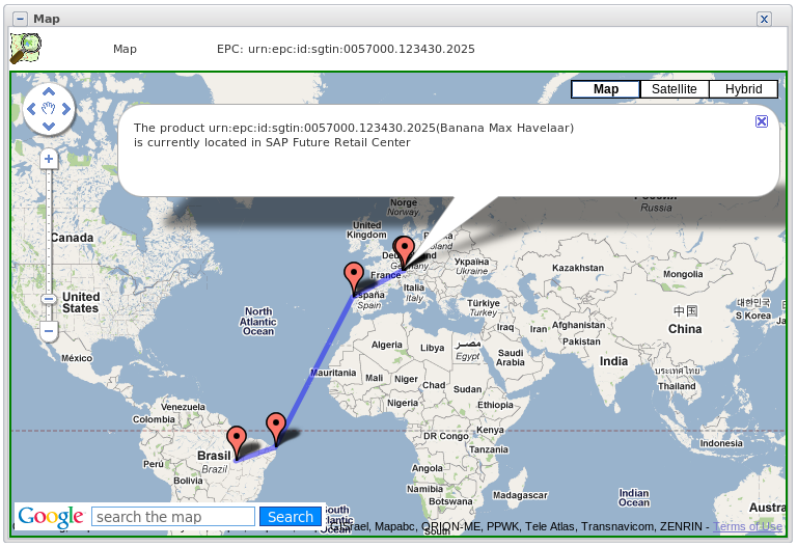

Fig. 8 The Maps widget is following the route of the banana tagged with the EPC urn:epc:id:sgtin:0057000.123430.2025

5.1.1 Mashup Architecture

The EPC Dashboard integrates several information sources. This information is encapsulated in small windows called widgets. The widgets combine services on the Web with traces coming from the RESTful EPCIS. The EPC Dashboard Mashup currently offers 12 widgets using different APIs and services. As an example, the Map Widget is built using the Google Maps Web API (see Figure 8), the Product Buzz Widget uses the Twitter RESTful API (Figure 7) and the Stock History Widget uses the Google Visualization API.

All widgets are connected to each other which means that actions on a given one can propagate the selection to the other widgets and changes their view accordingly. As such, widgets listen to selections and can make selections. This interaction is implemented using the observer pattern [4] where consumers (i.e., the widgets) register to asynchronous updates of the currently selected Locations, Readers, Time or EPCs. This architecture allows the creation and integration of other Web widgets with very little effort. The EPC Dashboard itself is a JavaScript application built using the Google Web Toolkit[13], a framework to develop rich Web clients. This has been possible because having a RESTful Interface upon the EPCIS which eases the development of mashups.

[13] http://code.google.com/intl/en/webtoolkit

6 Summary

In this chapter we argue that RESTful architecture can greatly contribute to the success of Information System. We further argue for thinking of these systems as Web APIs rather than as applications. As an illustration we describe how we applied the principles and constraints of RESTful architectures to the world of RFID for creating the RESTful EPCIS open-source project which is released as an open-source module of the Fosstrak project, under the name of epcis-restadapter[14].

RESTifying the EPCIS literally bring RFID traces to the Web, every tagged product, reader, location, etc. become fully addressable resources. Using the HTTP protocol tagged objects can be directly searched for, indexed, bookmarked, exchanged and feeds can be created by end-users. Furthermore, this enables exploring the EPCIS data simply by browsing them, which helps making sense of the data. We argue that this adds more flexibility to the types of applications that can be built on top of an EPCIS and opens the EPCIS API for fast-prototyping to the very large and active community of Web and mobile developers. We further show that this added flexibility does not necessarily have to hinder the overall performances, deploying the RESTful EPCIS on the same machine as the WS-* EPCIS leads to satisfactory results while preserving the EPCIS-vendor independence.

We finally illustrate the new application space the RESTful EPCIS unveils by means of a JavaScript Mashup: the EPC Dashboard which is an easily extensible business intelligence interface for managers that re-uses a number of Web APIs.

References

1. BILL BURKE. *RESTful Java with Jax-RS*. O'Reilly Media, 1st edition, November 2009.
2. ROY T. FIELDING and RICHARD N. TAYLOR. Principled design of the modern Web architecture. *ACM Trans. Internet Techn.*, 2(2):115–150, 2002.
3. CHRISTIAN FLOERKEMEIER, MATTHIAS LAMPE, and CHRISTOF RODUNER. Facilitating RFID Development with the Accada Prototyping Platform. In *Proceedings of the Fifth IEEE International Conference on Pervasive Computing and Communications Workshops*, pages 495–500. IEEE Computer Society, 2007.
4. ERICH GAMMA, RICHARD HELM, RALPH JOHNSON, and JOHN M. VLISSIDES. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, November 1994.
5. DOMINIQUE GUINARD, VLAD TRIFA, and ERIK WILDE. A Resource Oriented Architecture for the Web of Things. In *Proc. of IoT 2010 (IEEE International Conference on the Internet of Things)*, Tokyo, Japan, November 2010.
6. DOMINIQUE GUINARD, FELIX VON REISCHACH, and FLORIAN MICHAHELLES. MobileIoT Toolkit: Connecting the EPC Network to MobilePhones. In *Proc. of Mobile Interaction with the Real World at Mobile HCI (MIRW)*, Amsterdam, Netherlands, September 2008. The University of Oldenburg.
7. TIM KINDBERG, JOHN BARTON, JEFF MORGAN, GENE BECKER, DEBBIE CASWELL, PHILIPPE DEBATY, GITA GOPAL, MARCOS FRID, VENKY KRISHNAN, HOWARD MOR-

[14] http://www.webofthings.com/rfid

RIS, JOHN SCHETTINO, BILL SERRA, and MIRJANA SPASOJEVIC. People, places, things: web presence for the real world. *Mob. Netw. Appl.*, 7(5):365–376, 2002.

8. T. LUCKENBACH, P. GOBER, S. ARBANOWSKI, A. KOTSOPOULOS, and K. KIM. TinyREST - A protocol for integrating sensor networks into the internet. In *Proc. of the Workshop on Real-World Wireless Sensor Network (SICS)*, Stockholm, Sweden, 2005.
9. CESARE PAUTASSO and ERIK WILDE. Why is the Web Loosely Coupled? A Multi-Faceted Metric for Service Design. In *Proc. of the 18th International World Wide Web Conference (WWW'09)*, Madrid, Spain, April 2009.
10. CESARE PAUTASSO, OLAF ZIMMERMANN, and FRANK LEYMANN. Restful web services vs. big web services: making the right architectural decision. In *Proc. of the 17th international conference on World Wide Web (WWW)*, pages 805–814, New York, NY, USA, 2008. ACM.
11. LEONARD RICHARDSON and SAM RUBY. *RESTful Web Services*. O'Reilly Media, Inc., May 2007.
12. DOGAN YAZAR and ADAM DUNKELS. Efficient Application Integration in IP-based Sensor Networks. In *Proc. ACM of the First ACM Workshop On Embedded Sensing Systems For Energy-Efficiency In Buildings (BuildSys)*, Berkeley, CA, USA, November 2009.
13. JIN YU, BOUALEM BENATALLAH, FABIO CASATI, and FLORIAN DANIEL. Understanding Mashup Development. *IEEE Internet Computing*, 12(5):44–52, 2008.